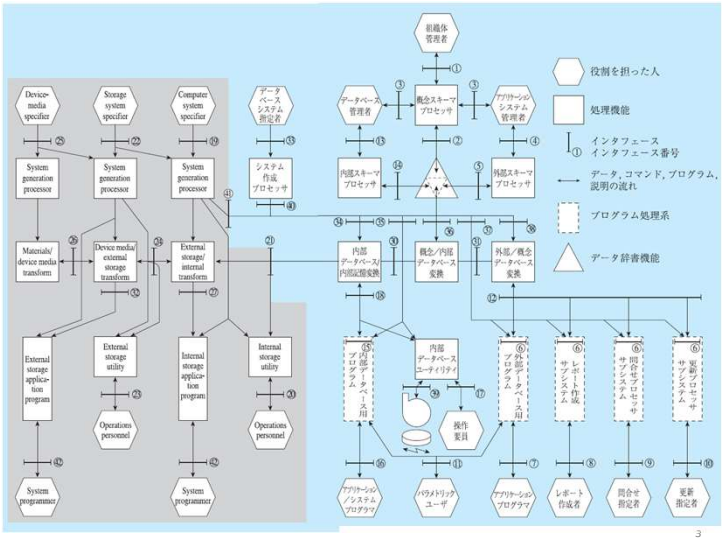
データベース第9回

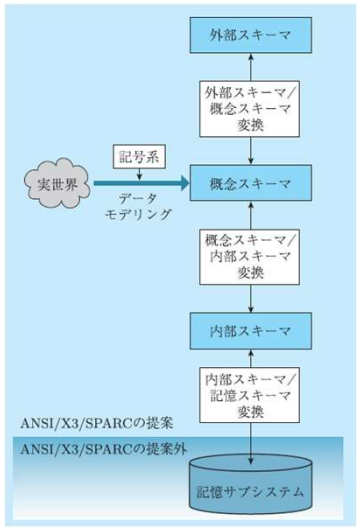
第9章　データベース管理システム

DBMSの標準アーキテクチャ

- 管理者
  - 組織体管理者
  - データベース管理者
  - アプリケーションシステム管理者
- 処理機能
  - 概念スキーマプロセッサ
  - 外部/概念データベース変換，概念/内部データベース変換，内部データベース/内部記憶変換
- データ辞書

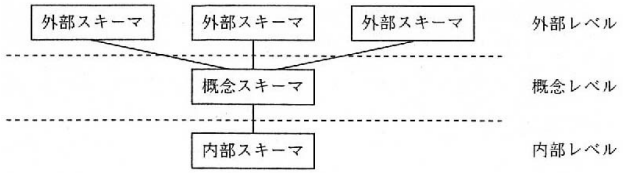
- 必要性が生じたときに見直してください





抽象化の3レベル

- DBMSの管理するスキーマの3階層
  - 外部スキーマ，概念スキーマ，内部スキーマ
  - ANSI/SPARCモデル[1975]（下図）

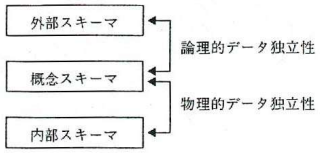


抽象化の3レベル

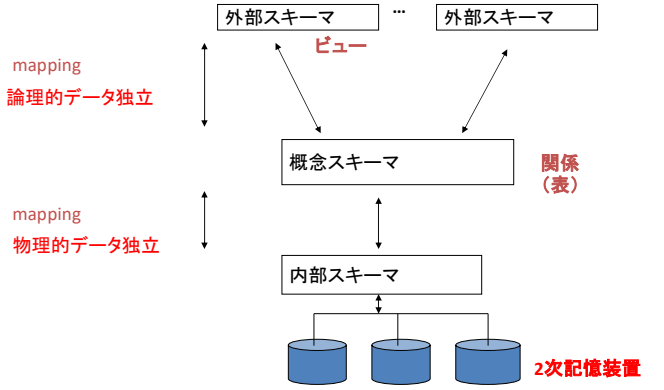
- DBMSの管理するスキーマの3階層
  - 内部スキーマ
    - 物理レベルのデータ構造
  - 概念スキーマ
    - データベースの定義言語で定義されるスキーマ
  - 外部スキーマ
    - 外部のアプリケーションから操作する時に，それぞれのアプリケーションインターフェイスに見せるスキーマ

データの独立性

- データベースではデータはアプリケーションとは独立して組織化される
  - 論理的データ独立性
    - 外部スキーマ･アプリケーションには影響を与えない範囲で概念スキーマを変更可能
  - 物理的データ独立性
    - 概念データスキーマに影響を与えない範囲で，内部スキーマ･物理データ構造を変更可能



DBMSの3層スキーマ構造



ビュー機能

- 概念スキーマ＝実リレーション
  - 実際に保存されているリレーション
  - （厳密には，保存されているように見えているだけで，内部スキーマ(＝物理構造)とは異なることが多い）
- 概念スキーマだけでは，データベースをアクセスするのに不便
- 概念スキーマから外部スキーマを生成．ユーザはそれを通してデータベースをアクセスする
- DBMSでは外部スキーマとしてビューがサポートされる

ビュー機能

- リレーショナルデータベースでは質問の結果がまたリレーションになるという性質を利用したもの
- ビューは，ビューにアクセスされるたびに実リレーションから計算（質問処理）され，ユーザに提供される
- ビューは仮想的なリレーション

選択ビュー

- 貧乏社員：リレーション社員（社員番号，社員名，所属，給与）から，給与が20未満の社員

```
CREATE VIEW 貧乏社員
AS
SELECT ＊
FROM 社員
WHERE 給与 < 20
```


結合ビュー

- 取引：供給（仕入先，部品）と需要（部品，納入先）を結合

```
CREATE VIEW 取引
AS
SELECT X.仕入先，Y.納入先
FROM 供給 X, 需要 Y
WHERE X.部品=Y.部品
```

ビューの利用

CREATE VIEW 貧乏社員
AS
SELECT ＊
FROM 社員
WHERE 給与 < 20

SELECT ＊
FROM 貧乏社員
WHERE 年齢 > 30

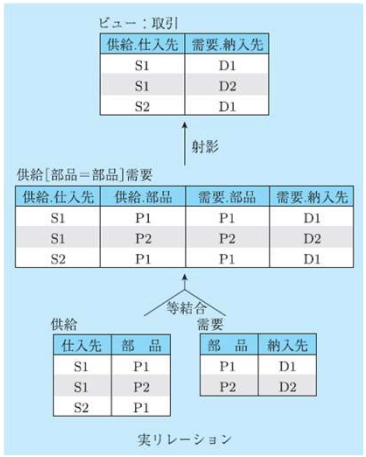
ビューの更新可能性

UPDATE 貧乏社員
SET 給与=給与×２

CREATE VIEW 貧乏社員
AS
SELECT ＊
FROM 社員
WHERE 給与<20

UPDATE 社員
SET 給与=給与×２
WHERE 給与<20

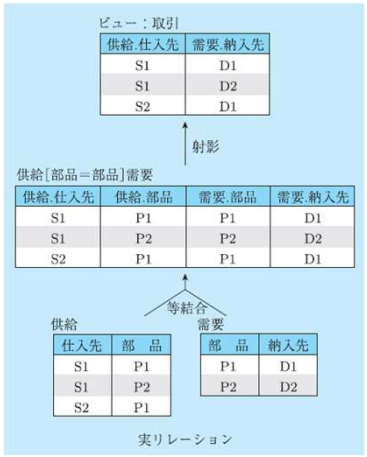
ビュー 取引



ビュー 取引から
(S1, D2)を削除

CREATE VIEW 取引
AS
SELECT X.仕入先, Y.納入先
FROM 供給 X, 需要 Y
WHERE X.部品=Y.部品

ビュー 取引



ビュー 取引から
(S1, D2)を削除

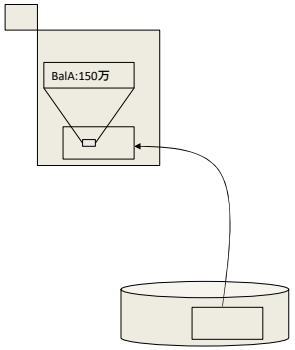
1. 供給から(S1,P2)を削除
2. 需要から(P2,D2)を削除
3. 上の1.，2. を両方

アクセスメソッド

- 大量のデータをリレーションとして保持
- なにかしら工夫をしておかないと，所望のデータにアクセス（検索や更新）する際，とてつもなく時間がかかることもある
- 一般的に，良く使われる属性（探索キーという）に対して，アクセスメソッドを用意する：インデックスを構築する：データベース管理者の仕事
- ここでは，B+木を紹介する

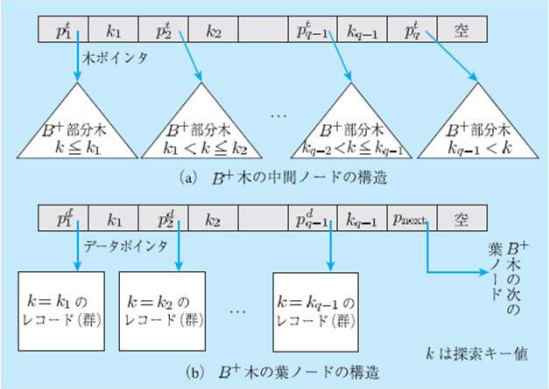
17

ディスクアクセス

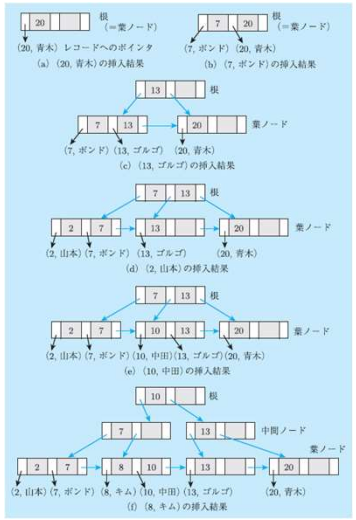


18

B+木のノード構造



19



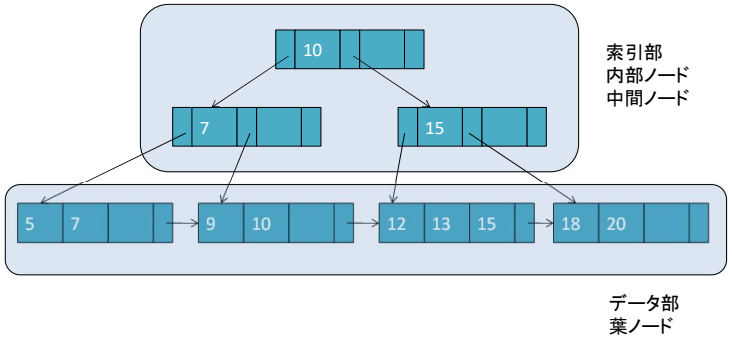
20

木構造インデックス

- インデックス構造内のデータの保持の仕方は以下の3通りある
  1. キー値kと一緒にデータそのものも保持
  2. キー値kと，kの値を持つレコードid（教科書でいうポインタ）を保持
  3. キー値kと，kの値を持つレコードidのリストを保持
- インデックスの1ノードを1ページで管理．これにより，ディスクI/O数を減らす．

21

B+木



22

B+木

- 最も広く使われているインデックス構造
- 平衡木（バランス木）：根から葉までの高さが等しい
- 挿入・削除は$\log_F N$で処理される．ここで，Fは分岐数(fanout)，Nは葉（ページ）の数
- 各ノードは，最低50%以上をデータ（キー値）で満たされてなければならない（根以外）
- 根以外の内部ノードのポインタは，上の項目のデータの個数＋1の数がなければならない
- 根は，少なくとも2個のポインタを持たなければならない
- 葉からのみデータを指し，葉は探索キーの値順に横並び
- 値検索と範囲検索をサポート

23

インデックスのアルゴリズム

- 検索
- 挿入
- 削除
- （バルクロード）

24

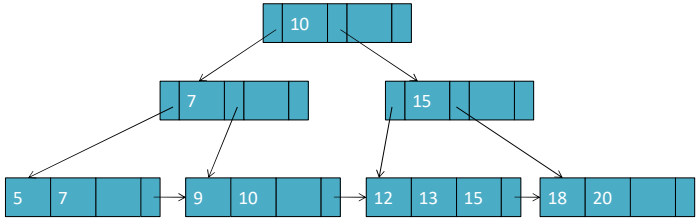
B+木への挿入アルゴリズム

- 正しい葉Lを見つける
- Lにデータを挿入する
  - Lに空きスペースがあったら，入れてお終い
  - なければ，LをLとL2に分割
    - L内のデータをL2と等分配し，中間の値(Lの中の最大値)を上位ノードにコピー
    - L2へのポインタをLの親ノードに挿入
- これを再帰的に行う
  - 内部ノードの分割には，ノード内のエントリを等分割し，中間の値を親ノードに押し上げる(コピーではない)
- ノードの分割により，木は大きくなる．ルートの分割は，木の高さが一つ増える．

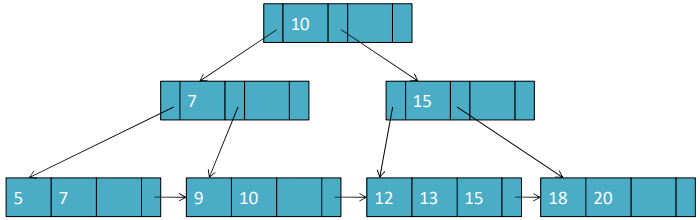
B+木からの削除アルゴリズム

- 削除するエントリがある葉Lを見つける
- エントリを削除
  - Lに少なくとも半分以上エントリがあれば，お終い
  - なければ
    - 隣接ノードからエントリを借りて，半分以上のエントリを持てるように試行する．必要なら，上位ノードにも更新を反映する．
    - 無理なら，Lと隣接ノードをマージ
- マージが起こったら，Lの親ノードからLか隣接ノードへのポインタを削除
- マージは，場合によってはルートまで生じ，結果高さが減ることもある

B+木

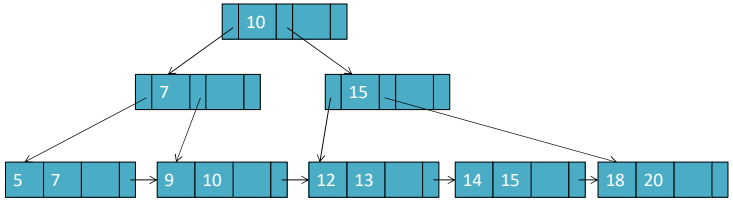


B+木



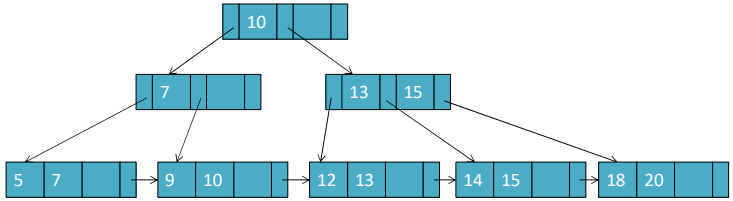
14の挿入？

14の挿入



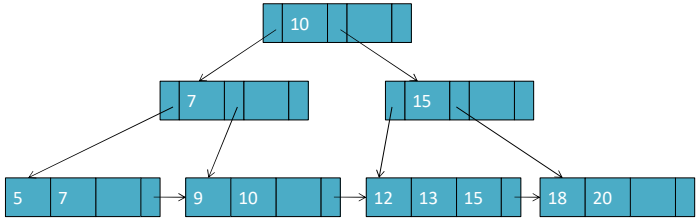
29

14の挿入　続き



30

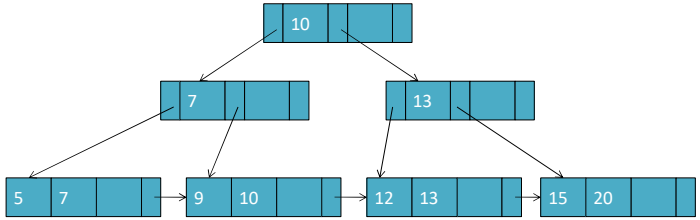
B+木



18の削除

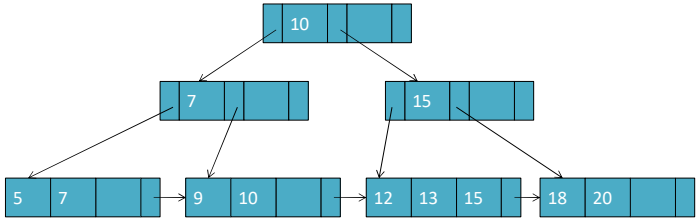
31

18の削除



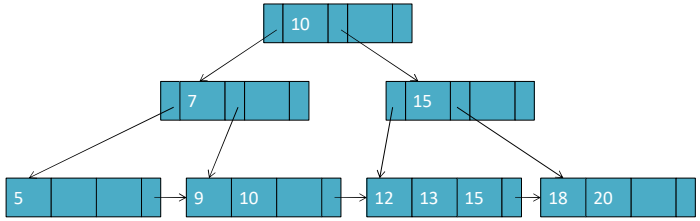
32

B+木

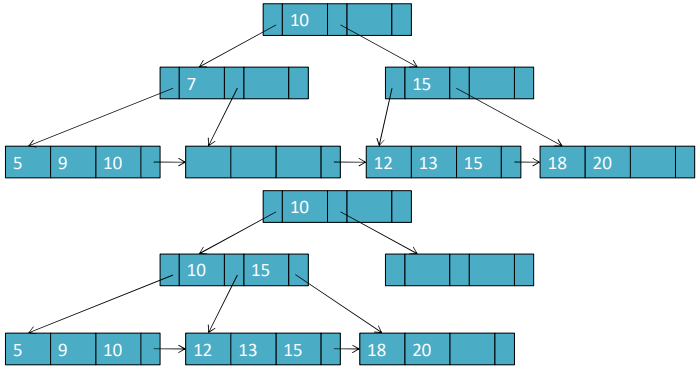


7の削除

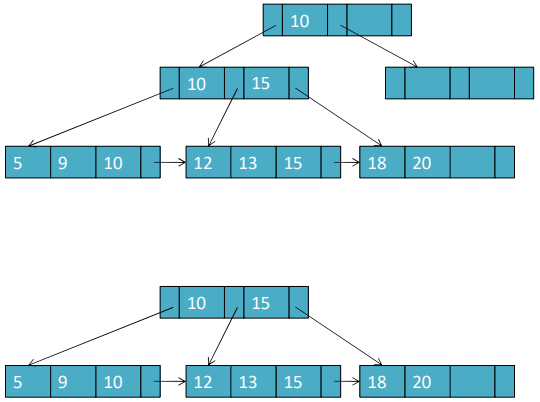
7の削除



7の削除 続き



7の削除 続き



B+木の詳細

- B+木でいくつのレコード(タップル)が管理されているか考えよう
- キー値が9バイト，ディスクブロックサイズが512バイト，レコードポインタが7バイト，ブロックポインタが6バイトとする
- 内部ノードが，多くてpブロックポインタを持つとすると…

37

B+木の詳細

- B+木でいくつのレコード(タップル)が管理されているか考えよう
- キー値が9バイト，ディスクブロックサイズが512バイト，レコードポインタが7バイト，ブロックポインタが6バイトとする
- 内部ノードが，多くてpブロックポインタを持つとすると…
- $(p \times 6)+((p-1) \times 9) \leq 512 \Rightarrow p=521/15=34$

38

B+木の詳細

- B+木でいくつのレコード(タップル)が管理されているか考えよう
- キー値が9バイト，ディスクブロックサイズが512バイト，レコードポインタが7バイト，ブロックポインタが6バイトとする
- 葉ノードが，多くてpレコードポインタを持つとすると…

39

B+木の詳細

- B+木でいくつのレコード(タップル)が管理されているか考えよう
- キー値が9バイト，ディスクブロックサイズが512バイト，レコードポインタが7バイト，ブロックポインタが6バイトとする
- 葉ノードが，多くてpレコードポインタを持つとすると…
- $(p \times (7+9))+6 \leq 512 \Rightarrow p=506/16=31$

40

B+木の詳細

- B+木でいくつのレコード(タップル)が管理されているか考えよう
- 各ノードが69%充足しているとする．高さが3だとすると…

B+木の詳細

- B+木でいくつのレコード(タップル)が管理されているか考えよう
- 各ノードが69%充足しているとする．高さが3だとすると
- Root: 1 node 22 entries 23 pointers
- Level 1: 23 nodes 506 entries 529 pointers
- Level 2: 529nodes 11,638 entries 12,167 pointers
- Leaf level: 12,167nodes 255,507 record pointers